2009 28th IEEE International Symposium on Reliable Distributed Systems

Relaxed Atomic Broadcast: State-Machine Replication Using Bounded Memory

Omid Shahmirzadi, Sergio Mena, André Schiper
École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
Email: {first.last}@epfl.ch

***Abstract*—Atomic broadcast is a useful abstraction for implementing fault-tolerant distributed applications such as state-machine replication. Although a number of algorithms solving atomic broadcast have been published, the problem of bounding the memory used by these algorithms has not been given the attention it deserves. It is indeed impossible to solve repeated atomic broadcast with bounded memory in a system (non-synchronous or not equipped with a perfect failure detector) in which consensus is solvable with bounded memory. The intuition behind this impossibility is the inability to safely garbage-collect unacknowledged messages, since a sender process cannot tell whether the destination process has crashed or is just slow.**

The usual technique to cope with this problem is to introduce a membership service, allowing the exclusion of a slow or silent process from the group and safely discarding unacknowledged messages sent to this process. In this paper, we present a novel solution that does not rely on a membership service. We *relax* the specification of atomic broadcast so that it can be implemented with bounded memory, while being strong enough to still be useful for applications that use atomic broadcast, e.g., state-machine replication.

I. Introduction and related work

Atomic broadcast has been proposed as the key abstraction to implement fault-tolerant distributed services [1] using the state-machine approach [2]. A number of different implementations of atomic broadcast have been proposed in the literature for a variety of system models [3]. However, they rarely tackle the problem of bounding the use of memory. The fact that an algorithm needs a potentially unbounded amount of buffers is often considered as a minor (implementation) issue. Bounding memory might not be a very exciting theoretical issue, it is nevertheless important from a practical point of view, since inability to bound (or garbage-collect) the memory used may lead to serious instability of the application, with effects similar to those of memory leaks. This is definitely not the best feature for algorithms that are supposed to increase availability. As Parnas argues in [4], a model should be simple, but if it becomes too simple it risks being a lie, i.e., not representing reality. No real system can assume it has access to unbounded memory.

Implementing atomic broadcast with bounded memory in a synchronous system is trivial [5]. However, if the system model does not allow us to distinguish a slow process from a crashed process, the ability of atomic broadcast algorithms to bound their memory – without affecting correctness – becomes challenging. Ricciardi [6] proved that a primitive as basic as (repeated) reliable broadcast cannot be implemented in a system with message losses in which slow processes are indistinguishable from crashed processes. Trivially, Ricciardi's impossibility result also applies to (repeated) atomic broadcast, since it is strictly stronger than (repeated) reliable broadcast. In this paper, we address the problem of bounded memory in the context of repeated atomic broadcast by weakening the specification of atomic broadcast. Note that (one instance of) consensus has been shown to be solvable with bounded memory [7] in an asynchronous system with the $\diamond\mathcal{S}$ failure detector, and in [8] Delporte-Gallet *et al.* show that solving (repeated) reliable broadcast requires indeed a stronger failure detector than solving (one instance of) consensus.

Group communication prototypes built in the last 20 years have addressed the problem of bounding memory thanks to group membership [9]–[12]: slow or irresponsive processes are excluded from the group so that messages sent to them can be safely garbage-collected before buffers at other processes overflow. However, this solution has its own drawbacks. First, the dynamic group model is more complex than the static one. Second, the dynamic model requires the introduction of a group membership service, which adds a performance overhead. Finally, excluding a destination process just because the sender is unable to garbage-collect its output buffers[1] may not always be desirable.

The paper presents *relaxed atomic broadcast*, a novel broadcast primitive defined in the static group model (i.e., no membership service), whose repeated invocation can be implemented using bounded memory. Relaxed atomic broadcast is weak enough so that it can be implemented with bounded memory, yet strong enough to be useful for applications that typically use atomic broadcast, such as state-machine replication. Note that repeated relaxed *atomic* broadcast is implementable with bounded memory in systems where repeated *reliable* broadcast is not. The intuition behind relaxed atomic broadcast is the following. As long as no process lags behind in the execution, relaxed atomic broadcast ensures the same properties as (classic) atomic broadcast. When some process p appears to be slow, other processes, instead of keeping on buffering messages for p, discard these messages. As a result, p will not be able to deliver all the messages that were atomically

Research funded by the Hasler Foundation under grant number 2070.

[1]This is called *output triggered suspicions* in [13].

broadcast. Missing messages are replaced at p with the special $\perp$ message (void), which signals that a message could not be delivered.

At first sight it may seem complicated, when using relaxed atomic broadcast for state-machine replication, to recover from the delivery of $\perp$. However, whenever some process p delivers $\perp$, the specification of relaxed atomic broadcast ensures that there exists some correct process that has delivered the missing message and applied it to its state. Thus state transfer, as in the case of dynamic groups, will allow p to recover from the delivery of $\perp$.[2]

The paper is organized as follows. The system model is presented in Section II. Section III discusses atomic broadcast and the problem of implementing repeated atomic broadcast with bounded memory. Approaches to address this are discussed in Section IV. Section V presents our novel approach. In Section VI, we present the implementation of relaxed atomic broadcast and its memory bounds. Section VII compares relaxed atomic broadcast over the solution that uses dynamic groups. Section VIII concludes the paper.

II. System model

We consider a system with a finite set of processes $\Pi = \{p_1, p_2, \dots, p_n\}$ that communicate by message exchange. Set Π has cardinality of n. We assume a partially synchronous system [14], where after some unknown time *GST* (*Global Stabilization Time*) the system (both processes and channels) becomes synchronous and channels become reliable.[3] Before GST the system is asynchronous and channels are lossy. Processes can only fail by crashing. A process that crashes stops its operation permanently and never recovers. A process is *faulty* in a run if it crashes in that run. A process is *correct* in a run if it is not faulty in that run. We only consider runs where up to f processes are faulty (f is a system parameter). Since processes do not know whether they are before or after GST, a slow process (or a process connected through a slow link) is indistinguishable from a crashed process.

Every pair of processes is connected by a bidirectional communication channel, which provides two communication primitives: *send*(m, q) and *receive*(m, q), where $m \in \mathcal{M}$ (the set of messages) and $q \in \Pi$. Channels satisfy the properties mentioned above. All messages in $\mathcal{M}$ are unique: they are broadcast at most once in a given run.

III. Repeated atomic broadcast and finite memory

We recall the definition of atomic broadcast. We say that a process p atomically broadcasts (or simply *abcasts*) message m if p executes *abcast*(m). Likewise, we say that a process p atomically delivers (or simply *adelivers*) message m if p executes *adeliver*(m). Atomic broadcast is defined by the following properties:

Property 3.1: Validity. If a correct process p abcasts message m, then some correct process eventually adelivers m.

Property 3.2: Uniform Integrity. Every process adelivers a message m at most once and only if m was previously abcast by some process.

Property 3.3: Uniform Agreement. If a process adelivers a message m then every correct process also adelivers m.

Property 3.4: Uniform Total Order. For any two processes p and q and any two messages m and m', if p adelivers m before m', then q adelivers m' only after having adelivered m.

Repeated atomic broadcast is the case where at least one process executes atomic broadcast infinitely often.

Reliable broadcast is defined by properties 3.1, 3.2, and the non-uniform version of 3.3. As shown by Ricciardi, repeated reliable broadcast cannot be implemented in a system with message losses in which slow processes are indistinguishable from crashed processes [6]. The intuition behind this impossibility result is the following. Consider a sender process p, and its output buffer to q that contains unacknowledged messages sent to q. If p is unable to distinguish whether q has crashed or is just slow (or connected through a slow link), then p cannot safely dispose of unacknowledged messages sent to q. However, if q has actually crashed, the set of unacknowledged messages will grow forever [13].

The impossibility of repeated reliable broadcast also applies to repeated atomic broadcast, since atomic broadcast is strictly stronger than reliable broadcast.

IV. How to deal with finite memory

Consider atomic broadcast used to implement state-machine replication [2] in a system with three processes ($n = 3$). Process p_1, which receives clients' requests, issues abcasts. Assume that the adelivery of these messages requires the cooperation of p_1 with only p_2 or with only p_3. Consider the former case, and assume p_3 is slow (or connected to p_1 and p_2 through slow channels). Since p_1 and p_2 do not know whether p_3 has crashed or not, they cannot safely dispose of unacknowledged messages sent to p_3, and their buffer to p_3 may grow infinitely.

We now present two approaches to deal with this problem.

The dynamic model: The traditional solution to bound memory consists in switching to the dynamic system (or dynamic group) model [9]–[12], [15].[4] In such a model processes can be added/removed to/from the system

[2]The size of the application state is controlled (and bounded) by the application. This is different from the state required for the implementation of atomic broadcast, which cannot be controlled by the application.

[3]We could also consider a system that alternates between sufficiently long *good* periods (system is synchronous and channels are reliable) and *bad* periods (system is asynchronous and channels are lossy). The algorithms would be the same.

[4]Note that this argument is not always explicit in these papers.

(or group) on the fly. In a dynamic model, a *view* describes the set of processes that are currently part of the system (or group). Views are maintained by a *membership* service, which adds and removes processes. Let us consider again state-machine replication with three replicas p_1, p_2 and p_3. If the buffer from p_1 to p_3 is full, p_1 may ask to remove p_3 from the view. Once this is done, all unacknowledged messages to p_3 can be discarded. However, the dynamic model is not so straightforward as the static one: protocol specifications and implementations have to be revised [16] and are more complex. Besides, a membership service is needed, and the application logic needs to become aware of view changes and state transfers (which are needed when an excluded process re-joins the group).

Relaxing the specification of atomic broadcast: The paper proposes another – novel – way to deal with bounded memory. Instead of switching to the dynamic model, we propose to *relax* the specification of atomic broadcast. This is done while keeping the specification strong enough to be useful for practical systems, and ensuring that repeated *relaxed* atomic broadcast is solvable with bounded memory.

V. Relaxed atomic broadcast

We start by defining relaxed atomic broadcast, and then we show how state-machine replication can be implemented using this new primitive.

A. Specification of relaxed atomic broadcast

We start by extending the set of messages that are delivered with the special void message $\perp$, which is not in set $\mathcal{M}$. Unlike any other message, this message is not unique, i.e., there may be more than one occurrence of this message in one run. A message m is called *normal* if it is not the void message $\perp$ (i.e., if $m \in \mathcal{M}$). The void message $\perp$ is never broadcast by the application, but might be delivered in substitution of a normal message in certain scenarios. The delivery of $\perp$ warns the application that a message is missing in its delivery sequence.

We define relaxed atomic broadcast with the primitives *xbcast(m)* and *xdeliver(m′)*, where $m \in \mathcal{M}$, and $m' \in \mathcal{M} \cup \{\perp\}$. Relaxed atomic broadcast is also called x-atomic broadcast. For k a positive integer, we say that a process p *xdelivers@*k message m if $m \in \mathcal{M} \cup \{\perp\}$ and m is the k^{th} message xdelivered by p since system start-up time. If k is irrelevant then the suffix @k is omitted, i.e., xdeliver@k simply becomes xdeliver. Relaxed atomic broadcast satisfies the following properties:

Property 5.1: VALIDITY. If a correct process $p \in \Pi$ xbcasts message m, then some correct process $q \in \Pi$ eventually xdelivers m.

This property does not change with respect to classic atomic broadcast (see Sect. III).

Property 5.2: UNIFORM AGREEMENT. For all $k \geq 1$, if some process xdelivers@k a normal message or $\perp$, then every correct process xdelivers@k a normal message or $\perp$.

The uniform agreement property is usually stated in terms of a given message m. In contrast, this weaker form only forces correct processes to xdeliver (at least) as many messages (normal or $\perp$) as any other process.

Property 5.3: UNIFORM TOTAL ORDER. For all $k, k' \geq 1$, if process p xdelivers@k normal message m and process q xdelivers@k' normal message m', then $k = k' \Leftrightarrow m = m'$.

The simplicity of the definition of uniform total order benefits from the definition of xdelivery@k. Property 3.4 could also benefit from this definition, thus becoming simpler.

Property 5.4: UNIFORM INTEGRITY. A process xdelivers a normal message m only if m was previously xbcast.

This property is simplified with respect to classic atomic broadcast for two reasons: (1) to allow the void message $\perp$ to be xdelivered more than once, and (2) because Property 5.3 already forbids xdelivering a normal message more than once.

Property 5.5: CONTINUITY. For all $k \geq 1$, a process xdelivers@k the void message $\perp$ only if at least one correct process xdelivers@k a normal message.

This safety property forbids runs where no correct process xdelivers a normal message at some position in the delivery sequence. Examples of such runs are (1) all processes xdeliver@k message $\perp$, or (2) correct processes xdeliver@k message $\perp$ and faulty processes xdeliver@k a normal message (and crash immediately after). In both cases, the application at surviving processes may not be able to reconstruct a complete delivery sequence of normal messages (i.e., without gaps).

The specification of relaxed atomic broadcast reduces to that of classic atomic broadcast in runs where no void message $\perp$ is ever xdelivered. Relaxed atomic broadcast is thus strictly weaker: any algorithm solving atomic broadcast also solves relaxed atomic broadcast.

B. Is the new specification useful?

We illustrate now the usefulness of relaxed atomic broadcast in the context of state-machine replication, see Algorithm 1. Basically, the algorithm works as though it was using classic atomic broadcast, but in addition it needs to implement a state transfer in order to recover from gaps in the sequence (when $\perp$ is xdelivered).

The (simple) algorithm works as follows. Two counters keep track of (1) the number of messages xdelivered, $n\text{-}xdel_p$; and (2) the number of (normal) messages that have been applied to the application's state, $n\text{-}st_p$ (i.e., $n\text{-}st_p$ messages, in sequence, have updated the application state). Initially these two counters match, and when a normal message is xdelivered both are incremented (lines 10 and 15).

If the void message $\perp$ is xdelivered, only $n\text{-}xdel_p$ is incremented to reflect the xdelivery, and process p halts its execution (line 13) until it receives a (more recent) state

Algorithm 1 State machine replication using relaxed atomic broadcast. Code for process p.

```
Initialization:
    $n\text{-}xdel_p \leftarrow 0$                {Number of messages xdelivered}
    $n\text{-}st_p \leftarrow 0$   {Number of messages applied to current state}
    $state_p \leftarrow$ initial state                {Replicated state}

task Main Thread
    repeat forever
        wait until received request $m$ from user
        xbcast($m$)

upon xdeliver($m$) do
    $n\text{-}xdel_p \leftarrow n\text{-}xdel_p + 1$
    if $n\text{-}xdel_p = n\text{-}st_p + 1$ then          {Any gaps so far?}
        if $m = \perp$ then
            wait until $n\text{-}xdel_p \leq n\text{-}st_p$
                        {Halt xdelivery of $\perp$ until a useful state received}
        else
            $n\text{-}st_p \leftarrow n\text{-}st_p + 1$
            $state_p \leftarrow$ apply $m$ to $state_p$

task Resend
    repeat forever
        if $n\text{-}xdel_p > n\text{-}st_p$ then
            send $\langle$STATE-REQ, $n\text{-}xdel_p\rangle$ to all

upon receive $\langle$STATE-REQ, $n\rangle$ from $q$ do
    if $n \leq n\text{-}st_p$ then
        send $\langle$STATE-REP, $n\text{-}st_p, state_p\rangle$ to $q$

upon receive $\langle$STATE-REP, $n, st\rangle$ from $q$ do
    if $n \geq n\text{-}xdel_p$ then
        $n\text{-}st_p \leftarrow n$
        $state_p \leftarrow st$
```

from another process q whose state has been updated by applying the message missing at p. To do so, if process p detects that the number of messages applied to its state ($n\text{-}st_p$) lags behind with respect to the number of messages xdelivered ($n\text{-}xdel_p$) due to the xdelivery of $\perp$, then p starts sending out state request messages repeatedly (line 20). When another process q receives the state request message (line 21), it checks whether its current state would be useful to the requesting process (the state is useful if it has been updated with at least as many messages as specified in the state request). If so, q sends back a state reply with its state and $n\text{-}st_p$. Finally, when the sender of the request receives a state reply (line 24) it checks whether that state is recent enough to fill the gaps in its xdelivery sequence. If it is the case, it replaces its state by the one it has just received, and updates $n\text{-}st_p$ accordingly. Note that the state received by p might have been updated with messages that have not (yet) been xdelivered at p. In this case, the algorithm ignores those messages when they are finally xdelivered (line 11).

If the application state is large, state transfer may be costly. However, this cost is the same as with the dynamic group solution.

Concurrency control: The state updates when an *upon* clause is executed should be atomic to avoid inconsistencies. A simple approach is to assume that the algorithm behaves like a monitor: *upon* clauses and tasks are executed in mutual exclusion, except when a *wait until* statement is reached, where another task or *upon* clause can take over the execution. Task *Resend* is an exception: it executes in mutual exclusion only within its loop: mutual exclusion is not preserved across consecutive executions of lines 19-20.

Memory bounds: The memory required by Algorithm 1 is bounded if we can bound the memory usage of relaxed atomic broadcast. Indeed, Algorithm 1 uses (1) two integers (M_{int} bits for each, see discussion in Section VI-B), (2) needs to store the application state that we assume to be bounded by M_{state} and a client request m that we assume to be bounded by M_{req}, and (3) needs memory space for the interaction between Algorithm 1 and the communication channels, and between Algorithm 1 and the relaxed atomic broadcast implementation (see Figure 1).

The interaction between Algorithm 1 and the communication channels is modeled by input and output buffers. Only one of each is represented in Figure 1, although we assume one pair for each channel (total of n pairs). Sending a message m is modeled by writing m into the output buffer. Receiving a message is modeled by an up-call that reads the input buffer and hands it over to Algorithm 1 (lines 21 and 24). These two buffers are bounded by the size of the longest message, the one with tag STATE-REP. The bound is $1 + M_{int} + M_{state}$ bits. The interaction between Algorithm 1 and the relaxed atomic broadcast implementation is modeled by function calls (xbcast is a down-call, xdeliver is an up-call). This interaction model does not add anything to the memory requirements of both components.

VI. Implementing repeated relaxed atomic broadcast with bounded memory

In this section, we present an algorithm that implements repeated relaxed atomic broadcast with bounded memory. For the sake of simplicity, from now on whenever we use the term relaxed atomic broadcast, we mean *repeated* relaxed atomic broadcast.

We first introduce the building blocks that our application (state machine replication) uses along with their interaction model, then we present the implementation of each building block followed by an analysis of the amount of memory needed. We will also have a short discussion regarding integers.

A. Building blocks and interaction model

Figure 1 depicts the building blocks of our implementation, as well as their interactions. Relaxed atomic broadcast uses consensus, and consensus is expressed in a round-based model implemented by the corresponding building block. The round-based model block interacts

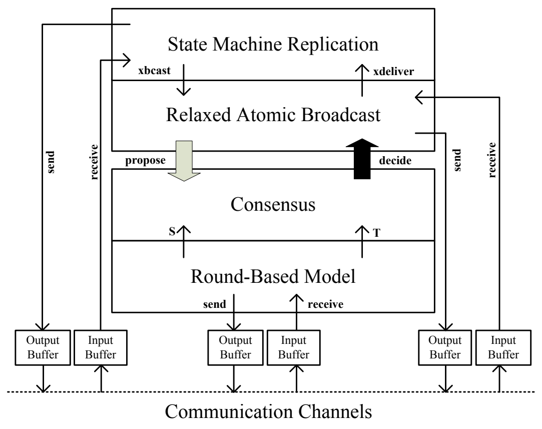

Figure 1. Building blocks. Small arrows represent function calls; large arrows represent spawning/killing (propose), and decision delivery (decide) of consensus instances.

with consensus by calling functions *S* and *T*. Likewise, state machine replication and relaxed atomic broadcast interact by calling functions *xbcast* and *xdeliver*, which are called in opposite direction. The interaction between relaxed atomic broadcast and consensus is different: when relaxed atomic broadcast calls *propose* a new instance of the consensus and round-based blocks (as well as their input/output buffers) is spawned, and any previous instance of these created blocks is immediately killed and garbage-collected. When consensus calls *decide*, a task within relaxed atomic broadcast is already waiting for it, so the call simply unblocks the task (and passes *decide*'s parameters) as we will see later. As explained above for state machine replication, the interaction with the channels is represented by input buffers and output buffers, one pair of buffers for the "relaxed atomic broadcast" block (i.e., one pair per channel), and one pair for the implementation of the round model (one pair per channel).

B. The issue of integers

Integer variables are used by all layers of our implementation. Some of these integers, such as message ids or round numbers are constantly increasing during system lifetime. This means that, at least theoretically, the number of bits needed by these variables cannot be bounded. However, this is not a problem from a practical point of view. Indeed, if we use 64 bits to represent some integer variable i, and we assume that i is increased by 1 every micro-second, then the largest integer is reached only after 584'000 years. This is long enough from a practical point of view (see also related discussion in [7]).

C. Relaxed atomic broadcast

1) Algorithm: Algorithm 2 solves relaxed atomic broadcast, for $f < n/2$, by reduction to a sequence of consensus [17]. However, contrary to [17], each consensus decides only on one single message (in order to bound memory) rather than on a batch of messages. Although a number of optimizations can be performed, we have kept the algorithm as simple as possible, while preserving its correctness (see [18] for the proofs).

The algorithm is structured in two tasks, *Sequencer* and *Gossip*, and works as follows. When p's application xbcasts a message m, a new identifier is attached to m. Then, m is stored in $Rcv_p[p]$ (line 9). Vector Rcv_p contains messages that p knows of but has not yet xdelivered. If p has previously xbcast another message m' not yet xdelivered, then p's application is blocked (i.e., p cannot xbcast any further message), since $Rcv_p[p]$ can only store one message at a time. This is a simple flow-control technique that can be optimized. The elements of vector Rcv_p will later become proposed values for consensus. This is the mission of task *Sequencer* (line 37), which executes a sequence of consensus instances. The *Sequencer* task waits until there are undelivered messages in vector Rcv_p (lines 39-40). Then, it starts a new consensus instance. For each instance #k_p a sender c_p is designated in a round-robin manner, with the goal to propose $Rcv_p[c_p]$ as the initial value for consensus (line 42). This initial value could be optimized to be the whole Rcv_p vector [17], but the rotating sender approach makes it easier to present both our algorithm and its memory bounds. When consensus #k_p decides, p waits for evidence that at least $f + 1$ other processes have also decided for consensus #k_p (line 45). This mechanism enforces the continuity property of relaxed atomic broadcast, since it ensures that at least one correct process (that can be queried later) has decided. Then, p xdelivers the message in $decision_p$ only if its identifier matches the value of $NextId_p[c_p]$, otherwise the decision is discarded (lines 46 and 29-32). This simple method demonstrates how to avoid xdelivering duplicates using bounded memory. Its side effect is that it enforces FIFO order amongst messages xbcast by process c_p. This may affect performance, but the algorithm can be optimized to relax this condition. Finally, variable k_p is incremented (line 33) and the loop starts over with a new iteration.

The *Gossip* task (line 34) sends periodically GOSSIP messages to all processes in order to disseminate (1) recently xbcast messages (vector Rcv_p), and (2) the status of the sender's current consensus instance (k_p and $decided_p$). When process p receives a GOSSIP message from process q (line 10), it checks whether q is either ahead or lagging behind. If q is ahead (or at the same consensus instance as p but has already decided), p adds q to its set $Finished_p$ (line 12), which contains processes that already finished p's current consensus. When the size of this set reaches $f + 1$, p can infer that at least one correct process has decided; so p can proceed to consensus k_p+1 as soon as it is done with consensus k_p (line 45 is no longer blocking). If q is lagging behind (line 13), then p simply sends q a SLOW message containing part of its current state. Additionally, if both p and q are at the same consensus instance (line 14), then p copies to its Rcv_p vector all messages received from q that p has not yet xdelivered.

A SLOW message conveys the part of the sender's state that a slow process needs in order to catch up. Upon

Algorithm 2 Solving relaxed atomic broadcast. Code for process p.

```
1: Initialization:
2:    $id_p \leftarrow 0$; $c_p \in \Pi$; $decision_p \in \mathcal{M}$
3:    $k_p \leftarrow 0$; $Finished_p \leftarrow \emptyset$; $decided_p \leftarrow$ false
4:    for all $r \in \Pi$ do $Rcv_p[r] \leftarrow \perp$; $NextId_p[r] \leftarrow 0$
5:    fork_task(Gossip, Sequencer)

6: upon xbcast($m$) do
7:    $m.id \leftarrow id_p$; $id_p \leftarrow id_p + 1$
8:    wait until $NextId_p[p] = m.id$
9:    $Rcv_p[p] \leftarrow m$

10: upon receive(GOSSIP, $k_q$, $d_q$, $Rcv_q$) from $q$ do
11:    if $k_q > k_p$ or $k_q = k_p$ and $d_q$ then
12:       $Finished_p \leftarrow Finished_p \cup \{q\}$
13:    if $k_q < k_p$ then send(SLOW, $k_p$, $NextId_p$) to $q$
14:    if $k_q = k_p$ then                              {Message dispersal}
15:       for all $r \in \Pi$ do
16:          if $Rcv_q[r] \neq \perp$
17:             and $Rcv_q[r].id = NextId_p[r]$ then
18:                $Rcv_p[r] \leftarrow Rcv_q[r]$

19: upon receive(SLOW, $k_q$, $N_q$) from $q$ do
20:    if $k_q > k_p$ then                                  {$p$ is late}
21:       kill_task(Sequencer)
22:       if $decided_p$ then deliver()
23:       $msgs\_skipped \leftarrow \sum_{r\in\Pi}(N_q[r] - NextId_p[r])$
24:       repeat $msgs\_skipped$ do xdeliver($\perp$)
25:       $NextId_p \leftarrow N_q$; $k_p \leftarrow k_q$
26:       $Finished_p \leftarrow \emptyset$; $decided_p \leftarrow$ false
27:       fork_task(Sequencer)

28: procedure deliver()
29:    if $decision_p \neq \perp$
30:       and $decision_p.id = NextId_p[c_p]$ then
31:          xdeliver($decision_p$)
32:          $NextId_p[c_p] \leftarrow NextId_p[c_p] + 1$
33:    $k_p \leftarrow k_p + 1$

34: task Gossip
35:    repeat forever
36:       send(GOSSIP, $k_p$, $decided_p$, $Rcv_p$) to all

37: task Sequencer
38:    repeat forever
39:       wait until $\exists r$ : ($Rcv_p[r] \neq \perp$
40:                      and $Rcv_p[r].id = NextId_p[r]$)
41:       $c_p \leftarrow k_p$ mod $|\Pi|$              {$c_p$ is a rotating sender}
42:       propose($k_p$, $Rcv_p[c_p]$)          {Delete previous instance}
43:       wait until decide($k_p$, $decision_p$)
44:       $decided_p \leftarrow$ true
45:       wait until $|Finished_p| > f$
46:       deliver()
47:       $Finished_p \leftarrow \emptyset$; $decided_p \leftarrow$ false
```

reception of such a message (line 19), process p checks whether the sender is ahead. If that is the case, p has been lagging behind, so termination of its current consensus instance is not guaranteed because other processes have already moved on to a later instance and disposed of p's current consensus (see Sect. VI-D). Therefore, p stops task *Sequencer* (line 21) and checks whether its current consensus had already finished. If so, the decision is xdelivered (line 22) and p advances to the next consensus. At this point, if p is still lagging behind with respect to q, the following catch-up mechanism is used. Process p calculates the number of messages it is going to skip when catching up: for each process r, p's next message id for process r is subtracted from q's (possibly greater) value (line 23). The result of this subtraction is the number of messages sent by r that were xdelivered between the consensus instances in which p and q are. The sum of all these subtractions yields the total amount of messages p will skip, so it xdelivers as many $\perp$ messages (line 24). Finally, p updates k_p and $NextId_p$ with the values received from q and spawns task *Sequencer* again. Note that additional garbage collection can be performed on Rcv_p, but does not affect correctness.

2) Concurrency control: The state of the protocol, in particular variables k_p, $NextId_p$, $Finished_p$, and $decided_p$, should all be updated atomically every time a new consensus instance starts. As in Sect. V-B, a simple approach is to assume that the algorithm behaves like a monitor: *upon* clauses and tasks are executed in mutual exclusion, except when a *wait until* statement is reached. Finally, task *Gossip* executes in mutual exclusion only within its loop (i.e., mutual exclusion is not preserved across consecutive executions of line 36).

3) Memory bounds: We show now that our algorithm requires only bounded memory as long as the size of the application payload is bounded to constant M_{req} (see Section V-B) and consensus requires a maximum of M_{cons} bits (see Section VI-D).

State size: To avoid a boring enumeration, let us assume that the space required for all variables except $decision_p$ and the vector Rcv_p amounts to some constant $c(n)$ (that depends on n). Moreover, $decision_p$ may contain an application message with an attached message id and vector Rcv_p is a vector of at most n application messages with added ids. Together this leads to $(n+1)\cdot(M_{req}+M_{int})$ bits. Since at most one consensus instance is running at each process, summing everything up, the state space needed by relaxed atomic broadcast is bounded by

$$M_{xbcast} = M_{cons} + (n+1)\cdot(M_{req} + M_{int}) + c(n).$$

Buffer size: The algorithm sends/receives two types of messages: GOSSIP and SLOW, with respectively four and three parameters. The former conveys the GOSSIP tag, one integer k_q, boolean d_q, and set Rcv_q of messages with attached ids. The latter contains the SLOW tag, one integer k_q, and set N_q of message ids. One bit is enough to represent the message tags. If we use again $c(n)$ to represent a constant depending on n, we get the following bounds:

$$M_{gossip} = n \cdot M_{req} + c(n),$$

$$M_{slow} = c(n).$$

D. Consensus

The relaxed atomic broadcast algorithm relies on a consensus algorithm, which ensures the following usual properties:

- *Validity:* If process p decides v, then v has been proposed by some process.
- *Uniform agreement:* No two processes decide differently.
- *Termination:* All correct processes eventually decide.

An unbounded number of consensus instances may be spawned in every run. Every instance of consensus uses its own memory resources. However, each process maintains only one single instance of consensus at a given time. When a process executes *propose*, its current consensus instance (if any) is immediately killed and garbage-collected. Therefore, the termination property of consensus is not guaranteed for all correct processes; rather, only $f+1$ processes (whether correct of not) are guaranteed to terminate a consensus instance. Nevertheless, once $f+1$ processes have decided for consensus #k (i.e., at least one correct process), Algorithm 2 guarantees that all correct processes will eventually stop consensus #k and move on to #$k+1$.

1) Algorithm:

Round-based model: We consider a consensus algorithm for a partially synchronous system (see Section II). As in [14], we consider an abstraction on top of the system model, namely a round model. Using this abstraction, rather than the raw system model, improves the clarity of the algorithms and simplifies the proofs. In the round model, processing is divided into rounds of message exchange. Each round r consists of a *sending step* denoted by S_p^r (sending step of p for round r), and of a *state transition step* denoted by T_p^r. In a sending step, each process sends a message to all. A subset of the messages sent is received at the beginning of the state transition step: messages can get lost, and a message sent in round r can only be received in round r. We denote by σ_p^r the message sent by p in round r, and by $\vec{\mu}_p^r$ the messages received by process p in round r ($\vec{\mu}_p^r$ is a vector of size n, where $\vec{\mu}_p^r[q]$ is the message received from q or *null* if the message was lost). Based on $\vec{\mu}_p^r$, process p updates its state in the state transition step.

In all rounds executed before GST messages can be lost. However, after GST, there exists a round GSR (*Global Stabilization Round*) such that the message sent in round $r \geq GSR$ by a correct process q to a correct process p is received by p in round r. This is formally expressed by the following predicate (where $\mathcal{C}$ denotes the set of correct processes):

$$\forall r \geq GSR : \mathcal{P}_{good}(r),$$

where

$$\mathcal{P}_{good}(r) \equiv \forall p, q \in \mathcal{C} : \ \vec{\mu}_p^r[q] = \sigma_q^r.$$

An algorithm that ensures this predicate in a partially synchronous system is given in Section VI-E.

Algorithm 3 The *OneThirdRule* (OTR) algorithm [19] ($f < n/3$). Code for process p.

```
1:  Initialization:
2:      $x_p \leftarrow v_p$

3:  Round $r$:
4:      $S_p^r$:
5:          send $\langle x_p \rangle$ to all processes

6:      $T_p^r$:
7:          if (number of messages sent in round $r$ and
            received by $p$ in round $r$)$> 2n/3$ then
8:              if the values received, except at most $\lfloor \frac{n}{3} \rfloor$, are
                equal to $\bar{x}$ then
9:                  $x_p \leftarrow \bar{x}$
10:             else
11:                 $x_p \leftarrow$ smallest $x$ received
12:             if more than $2n/3$ values received are equal to
                $\bar{x}$ then
13:                 DECIDE($\bar{x}$)
```

The OTR consensus algorithm: Algorithm 3 is the consensus algorithm we consider [19]. The algorithm requires $f < n/3$. We have chosen this algorithm because of its simplicity. The analysis of *Paxos/LastVoting* [19], [20], which requires only $f < n/2$ could be used instead, but would require more space.

Algorithm 3 works as follows. As soon as more than $2n/3$ processes have $x_p = v$, then decision v is locked, i.e., in any future update, variable x_p, is updated to v. Termination is ensured by the following observation. Let r_0 be the smallest round after GSR such that all faulty processes have crashed before round r_0. In round r_0 the condition of line 7 is true. Moreover, $\mathcal{P}_{good}(r_0)$ ensures that all processes that execute round r_0 receive the same set of messages. Therefore, in round r_0, either all processes execute line 9, or all processes execute line 11. It follows that at the end of round r_0 all processes have x_p equal to some common value v, and all processes decide in round $r_0 + 1$.

2) Memory bounds:

As we explain in Section VI-E, the memory required by Algorithm 3 is managed by the implementation of the round-based model. Thus we refer to the next section for the consensus memory bounds.

E. Implementation of the round-based model

We describe now the implementation of the round-based model (see Algorithm 4), which is almost identical to the one appearing in [21] (we made small extensions to prevent $msgsRcv_p$ from growing forever). The interaction between Algorithm 4 and Algorithm 3 is by function call: in other words, the execution thread is within Algorithm 4, and this thread calls functions S_p^r and T_p^r defined by Algorithm 3:

- S_p^r is called at line 9 of Algorithm 4 and returns x_p, see line 5 of Algorithm 3.[5]
- T_p^r is called at line 22 of Algorithm 4 and returns the new state of process p, see lines 7 to 13 of Algorithm 3.[6]

The state of Algorithm 3 is represented as s_p in Algorithm 4 (line 3). Moreover, in Algorithm 4, ϕ represents the bound on process relative speed after GSR, and δ represents the bound on message transmission delay after GSR. After GSR one send step (line 10) and one receive step (line 16) take each 1 time unit on the fastest process (i.e., at most ϕ time units on the slowest process). If no message is available for reception, then an empty message is received. In one send step a process can send messages to multiple processes, while n receive steps are needed to receive messages from n processes.

1) Algorithm: Algorithm 4 consists of an infinite loop (see line 8), which includes an inner loop (lines 12 to 21). Each iteration of the outer loop corresponds to one round. The message to send is obtained in line 9, and sent to all in line 10. Each iteration of the inner loop is for the reception of one message for the current round r_p. The inner loop ends when (i) at least $2\delta+(n+2)\phi$ time units have elapsed, see lines 14-15 (time is measured by the execution of receive steps: 1 receive step = 1 time unit), or (ii) whenever a message of a round larger than r_p is received, see lines 20-21. The reader is referred to [21] for a proof that this ensures $\mathcal{P}_{good}$ after GST. When the inner loop ends, the function T_p^r is called with the set of messages received in the current round r_p (line 22). Finally, messages for the current round are garbage collected (line 24).

2) Memory bounds: We compute now M_{cons} – the memory bound for consensus including the implementation of the round-based model – that was referenced in Section VI-C3.

State size: Algorithm 4 needs to store three integers (r_p, $next\_r_p$, i_p), which require $3M_{int}$, and variables s_p and msg_p, which require $2M_{req}$ bits. In addition the algorithm needs memory for $msgsRcv_p$ and $temp_p$, which amounts to $(n+1)\cdot(M_{req}+2M_{int})$ bits, since $msgsRcv_p$ stores at most n messages.

Buffer size: All messages sent/received are of the same type and require at most $M_{req} + M_{int}$ bits each. The algorithm needs only one single output buffer (the same message sent to all) and n input buffers (one per process). This amounts to $(n+1)\cdot(M_{req}+M_{int})$ bits.

F. Summary

Putting everything together, we have shown that all components that appear in Figure 1, including state-machine replication, require only bounded memory. Therefore, relaxed atomic broadcast has allowed us to implement state-machine replication using bounded memory.

[5]To be consistent, line 4 of Algorithm 3 should be expressed as a function. However, we decided to keep the usual round-based expression for Algorithm 3.

[6]Same comment as for S_p^r, see footnote 5.

Algorithm 4 Ensuring $\mathcal{P}_{good}$ after GST.

```
1: $r_p \leftarrow 1$                              {round number}
2: $next\_r_p \leftarrow 1$                        {next round number}
3: $s_p \leftarrow init_p$                         {state of the consensus algorithm}
4: $i_p \leftarrow 0$                              {counts send/receive steps}
5: $msg_p$                                         {message to send in the current round}
6: $msgsRcv_p \leftarrow \emptyset$                {set of msgs received for the current round}
7: $temp_p \leftarrow \emptyset$                   {contains at most one message received for a round $> r_p$}
8: while true do
9:     $msg_p \leftarrow S_p^{r_p}(s_p)$
10:    send $\langle msg_p, r_p \rangle$ to all
11:    $i_p \leftarrow 0$
12:    while $next\_r_p = r_p$ do
13:        $i_p \leftarrow i_p + 1$
14:        if $i_p \geq 2\delta + (n+2)\phi$ then
15:            $next\_r_p \leftarrow r_p + 1$
16:        receive a message with highest round number
17:        if received $\langle msg, r' \rangle$ from $q$ then
18:            if $r' = r_p$ then
19:                $msgsRcv_p \leftarrow msgsRcv_p \cup \{\langle msg, r', q \rangle\}$
                   {Messages from old rounds are discarded}
20:            if $r' > r_p$ then
21:                $next\_r_p \leftarrow r'$;  $temp_p \leftarrow \{\langle msg, r', q \rangle\}$
22:    $s_p \leftarrow T_p^{r_p}(msgsRcv_p, s_p)$
23:    $r_p \leftarrow next\_r_p$
24:    $msgRcv_p \leftarrow temp_p$                {Garbage collection}
25:    $temp_p \leftarrow \emptyset$
```

VII. Comparison of approaches

In Section IV, we have presented two different approaches for implementing state machine replication with bounded memory. Namely, (1) our novel relaxed atomic broadcast algorithm, which was described in detail in Sections V and VI, and (2) atomic broadcast in the dynamic model, i.e., relying on membership [15]. Both approaches rely on state transfer: approach (2) requires a state transfer whenever a new process is added to the dynamic group; approach (1) performs a state transfer whenever a slow process catches up.

Solution (2) is more complex than solution (1). First, solution (2) needs to define a policy for process exclusion [22]. This is simply not needed in (1). Second, static group communication is simpler and easier to understand than dynamic group communication, from a specification as well as from an implementation point of view. Moreover, the complexity added by relaxed atomic broadcast (i.e., the need for state transfer) is also needed in dynamic group communication, as stated above.

If an application is happy with the static group model, and dynamism is introduced only to bound the memory usage, then the solution using relaxed atomic broadcast is a better choice. If an application requires the dynamic group model, the solution using relaxed atomic broadcast may still be used: it makes sense to combine both approaches, where changes in membership are decoupled from the bounded memory issue.

VIII. Conclusion

We have presented relaxed atomic broadcast, a variant of atomic broadcast that it is weak enough to be solved with bounded memory, yet strong enough to be useful for typical applications like state machine replication. Note that the analysis of the memory requirements forced us to consider the complete protocol stack (i.e., nothing has been swept under the carpet). We have also discussed the advantages of our approach as compared to the solution with group membership.

The solution presented shows an interesting trade-off between the memory allocated and the number of $\perp$ messages delivered: if a process becomes slow, the more memory we allocate, the longer it will take to run out of buffers. We plan to experimentally analyze this trade-off in the future.

Finally, we recall that the goal when presenting our solution was simplicity. The algorithm can be optimized in a number of ways in order to improve its performance.

References

[1] K. P. Birman, "Replication and fault-tolerance in the Isis system," in *Proceedings of the 10th ACM Symp. on Operating Systems Principles (SoSP-10)*, vol. 19, no. 5. Orcas Island, WA, USA: ACM, Dec. 1985, pp. 79–86.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[3] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2005.

[4] D. L. Parnas, "Use the simplest model, but not too simple," *Forum, Commun. ACM*, vol. 50, no. 6, p. 7, 2007.

[5] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[6] A. Ricciardi, "Impossibility of (repeated) reliable broadcast," Univ of Texas, Austin, Tech. Rep. TR-PDS-1996-003, April 1996.

[7] R. Guerraoui, R. Oliveira, and A. Schiper, "Stubborn communication channels," École Polytechnique Fédérale de Lausanne, Switzerland, Tech. Rep. 98/272, Mar. 1998.

[8] C. Delporte-Gallet, S. Devismes, H. Fauconnier, F. Petit, and S. Toueg, "With finite memory consensus is easier than reliable broadcast," in *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 41–57.

[9] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: a communication sub-system for high availability," in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, Boston, MA, USA, Jul. 1992, pp. 76–84.

[10] R. Macêdo, P. D. Ezhilchelvan, and S. K. Shrivastava, "Flow control schemes for a fault-tolerant multicast protocol," ESPRIT Basic Research Project BROADCAST, Technical Report BROADCAST-TR95-91, Jun. 1995.

[11] S. Mishra and L. Wu, "An evaluation of flow control in group communication," *IEEE/ACM Trans. Netw.*, vol. 6, no. 5, pp. 571–587, 1998.

[12] M. Hayden, "The Ensemble system," Ph.D. dissertation, Cornell University, Ithaca, NY, 1998.

[13] B. Charron-Bost, X. Défago, and A. Schiper, "Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently," in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Osaka, Japan, Oct. 2002, pp. 244–249.

[14] C. Dwork, N. A. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[15] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

[16] A. Schiper, "Dynamic group communication," *Distributed Computing*, vol. 18, no. 5, pp. 359–374, 2006.

[17] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[18] O. Shahmirzadi, S. Mena, and A. Schiper, "Relaxed atomic broadcast: State-machine replication using bounded memory," École Polytechnique Fédérale de Lausanne, Switzerland, Tech. Rep. LSR/2009/02, Jun. 2009.

[19] B. Charron-Bost and A. Schiper, "The Heard-Of Model: Computing in Distributed Systems with Benign Failures," École Polytechnique Fédérale de Lausanne, Switzerland, Tech. Rep. LSR/2007/01, Jul. 2007, to appear in Distributed Computing.

[20] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[21] M. Hutle and A. Schiper, "Communication predicates: A high-level abstraction for coping with transient and dynamic faults," *Proc. of Int. Conference on Dependable Systems and Networks (DSN'07)*, vol. 00, pp. 92–101, Jun. 2007.

[22] A. Schiper and S. Toueg, "From Set Membership to Group Membership: A Separation of Concerns," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 2, pp. 2–12, 2006.